Introduction to React.js

Trayan Iliev

IPT – Intellectual Products & Technologies
e-mail: tiliev@iproduct.org
web: http://www.iproduct.org

Agenda

1. MVC flavours
2. Singe Page Applications (SPA)
3. SIMPLE Webpack Project Bootstraping
4. Why React - simple and superfast, component oriented development using pure JavaScript (ES 6), virtual DOM, one-way reactive data flow, MVC framework agnostic
5. React by example – JSX syntax
6. React by example – JavaScript syntax
7. Lets do some code :)
8. Top level API
9. ES6 class syntax

Agenda

10. JSX in depth – differences with HTML, transformation to JavaScript, namespaced components,
11. Expressions, child expressions and comments, props mutation anti-pattern, spread attributes, using HTML entities, custom attributes, if-else, immediately-invoked function expressions.
12. React Components Lifecycle Callbacks and ES6 class syntax
13. Events in React, managing DOM events
14. Components composition in depth – ownership, *this.props.children*, *React.Children* utilities, child reconciliation, stateful children and dynamic children using keys
15. Transferring props

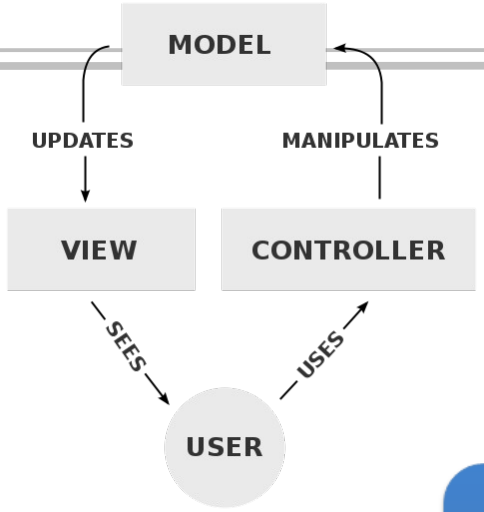
MVC Comes in Different Flavors

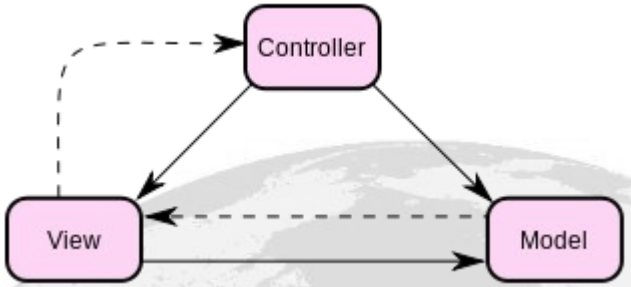
What is the difference between following patterns:

- Model-View-Controller (MVC)
- Model-View-ViewModel (MVVM)
- Model-View-Presenter (MVP)

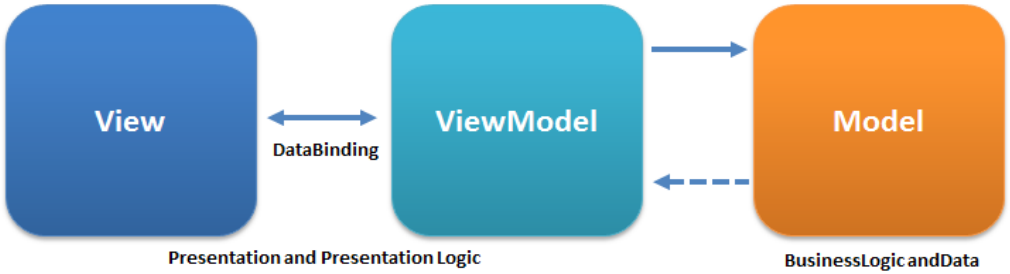
MVC Comes in Different Flavors - 2

- MVC

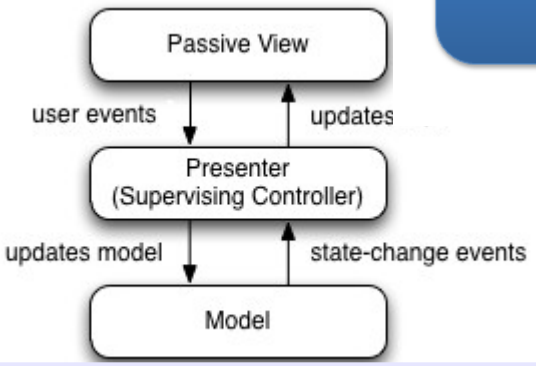




- MVVM

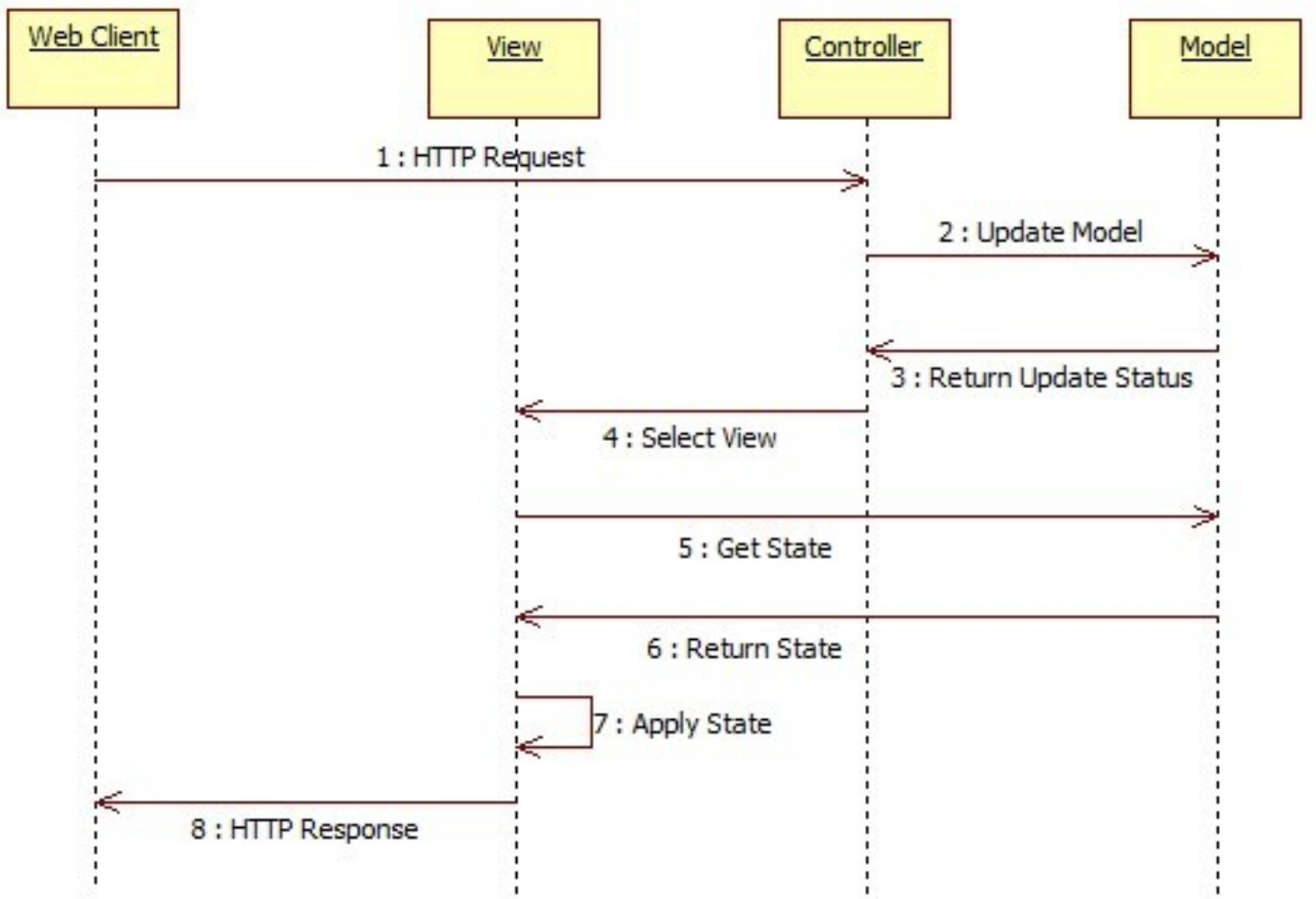


- MVP

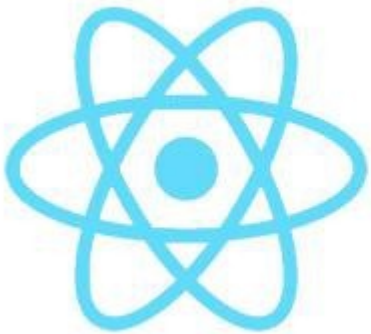


Sources:https://en.wikipedia.org/wiki/Model_View_ViewModel#/media/File:MVVMPattern.png, https://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93presenter#/media/File:Model_View_Presenter_GUI_Design_Pattern.png
License: CC BY-SA 3.0, Authors:Ugaya40, Daniel.Cardenas

Web MVC Interactions Sequence Diagram

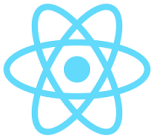


Why React?



React

- **React.js** is a JavaScript library for creating user interfaces by Facebook and Instagram – the V in MVC.
- **Solves well one problem**: building large applications with data that changes over time
- Simple and **superfast** – one-way reactive data flow



Why React?

- **Declarative** and **one-way reactive data flow** – simply express how your app should look, and React will automatically manage all UI updates when your underlying data changes
- **Component oriented SPA** development using pure JavaScript (**ES 6**) – React is all about building composable and reusable components - code reuse, testing, and separation of concerns
- **Virtual DOM** – allows decoupling of components from DOM, rendering done as last step
- Allows **isomorphic (client + server side) rendering**
- **MVC framework agnostic** – Flux, Redux, Reflux, ...
- Availabale at: ***https://facebook.github.io/react***

React.js by Example – JSX Syntax

```
import React from "react";
import ReactDOM from "react-dom";
import Hello from "./hello";

ReactDOM.render(
  <Hello name="World" />,
  document.getElementById('app')
);
```

JavaScript syntax extension (JSX) that looks similar to XML

```
import React from "react";

export default
class Hello extends React.Component
{
  render() {
    return (
      <div className="hello">
        <h2>
          Hello, {this.props.name}!
        </h2>
      </div>
    );
  }
}
```

React.js by Example – Older Version

```
import React from "react";
import ReactDOM from "react-dom";
import Hello from "./hello";

ReactDOM.render(
  <Hello name='World' />,
  document.getElementById('app')
);
```

JavaScript syntax extension (JSX) that looks similar to XML

```
import React from "react";

export const Hello =
React.createClass({
  render: function() {
   return (
     <div>
         Hello, {this.props.name}!
     </div>
   );
  },
});
```

Hello React Example + Property Type Validation

```
import React from "react";

export const Hello = React.createClass({
  propTypes: {
    name: React.PropTypes.string
  },
  render: function() {
   return (
     <div>
         Hello, {this.props.name}!
     </div>
   );
 },
});
```

Comments Demo Example – Pure JavaScript

```
import React from "react";
import ReactDOM from "react-dom";

let CommentBox = React.createClass({displayName: 'CommentBox',
  render: function() {
    return (
      React.createElement('div', {className: "commentBox"},
        "Hello, world! I am new  CommentBox."
      )
    );
  }
});

ReactDOM.render(
  React.createElement(CommentBox, null),
  document.getElementById('app')
);
```

Lets Do Some React Code :)

Official React Comments tutorial:

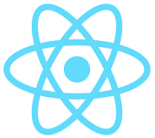
https://facebook.github.io/react/docs/tutorial.html

React comment box example available @GitHub:

https://github.com/reactjs/react-tutorial

React.js documentation and API:

https://facebook.github.io/react/docs



Top Level API

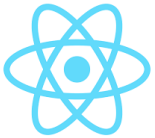
- **React** – the entry point to the React library. If you're using one of the prebuilt packages it's available as a global; if you're using CommonJS modules you can require() it.
- **ReactDOM** – provides DOM-specific methods that can be used at the top level of your app and as an escape hatch to get outside of the React model if you need to. Most of your components should not need to use this module.
- **ReactDOMServer** – the react-dom/server package allows you to render your components on the server: **ReactDOMServer.renderToString(ReactElement element)**
- @: ***https://facebook.github.io/react/docs/top-level-api.html***

React ES6 Demo Example

```
export class Counter extends React.Component {
  constructor(props) {
    super(props);
    this.state = {count: props.initialCount};
    this.tick = this.tick.bind(this);
  }
  tick() {
    this.setState({count: this.state.count + 1});
  }
  render() {
    return (
      <div onClick={this.tick}>
        Clicks: {this.state.count}
      </div>
    );
  }
}
Counter.propTypes = { initialCount: React.PropTypes.number };
Counter.defaultProps = { initialCount: 0 };
```

Agenda

1. Let's start with some code
2. JSX in depth – differences with HTML, transformation to JavaScript, namespaced components,
3. Expressions, child expressions and comments, props mutation anti-pattern, spread attributes, using HTML entities, custom attributes, if-else, immediately-invoked function expressions.
4. React Components Lifecycle Callbacks and ES6 class syntax
5. Events in React, managing DOM events
6. Components composition in depth – ownership, *this.props.children*, *React.Children* utilities, child reconciliation, stateful children and dynamic children using keys
7. Transferring props



Top Level API

- **React** – the entry point to the React library. If you're using one of the prebuilt packages it's available as a global; if you're using CommonJS modules you can require() it.
- **ReactDOM** – provides DOM-specific methods that can be used at the top level of your app and as an escape hatch to get outside of the React model if you need to. Most of your components should not need to use this module.
- **ReactDOMServer** – the react-dom/server package allows you to render your components on the server: **ReactDOMServer.renderToString(ReactElement element)**
- @: ***https://facebook.github.io/react/docs/top-level-api.html***

React Components

- **Virtual DOM** – everything is a component (e.g. in JSX), rendering done as last step
- **Components are like functions** – of three arguments:
- **this.props** – these is the external interface of the component, passed as attributes – allow the parent component ("owner") to pass state and behavior to embedded ("owned") components. Should never be mutated within component – immutable.
- **this.props.children** – part of the component interface but passed in the body of the component (component tag)
- **this.state** – internal state of the component should be mutated only using **React.Component.setState(nextState, [callback])**

Comments Demo Example – JSX

```
import React from "react";

export const Hello = React.createClass({
  propTypes: {
    name: React.PropTypes.string
  },
  render: function() {
   return (
     <div>
         Hello, {this.props.name}!
     </div>
   );
 },
});
```

React ES6 Demo Example

```
export class Counter extends React.Component {
  constructor(props) {
    super(props);
    this.state = {count: props.initialCount};
    this.tick = this.tick.bind(this);
  }
  tick() {
    this.setState({count: this.state.count + 1});
  }
  render() {
    return (
      <div onClick={this.tick}>
        Clicks: {this.state.count}
      </div>
    );
}}
Counter.propTypes = { initialCount: React.PropTypes.number };
Counter.defaultProps = { initialCount: 0 };
```

Stateless Components as Pure Functions

```
function HelloMessage(props) {
    return <div>Hello {props.name}</div>;
}
ReactDOM.render(<HelloMessage name="React User" />,
mountNode);
```

- OR using ES6 => syntax:

```
const HelloMessage = (props) => <div>Hello {props.name}</div>;
ReactDOM.render(<HelloMessage name="React User" />,
mountNode);
```

What Components Should Have State?

- Most components should just render data from props. However, sometimes you need to **respond to user input**, a **server request** or the **passage of time** => then use state.
- Try to keep as many of your components as possible **stateless** – makes easier to reason about your application
- Common pattern: create several **stateless components** that just **render data**, and have a **stateful component above them** in the hierarchy that **passes its state to its children via props**
- **Stateful component** encapsulates all of the **interaction logic**
- **Stateless components** take care of **rendering data in a declarative way**

JSX Syntax

- With JSX: Hello!
- In pure JS: React.createElement('a',
  {href: 'https://facebook.github.io/react/'}, 'Hello!')

- JSX is optional – we could write everything without it:

```
var child1 = React.createElement('li', null, 'First Text Content');
var child2 = React.createElement('li', null, 'Second Text Content');
var root = React.createElement('ul', { className: 'my-list' }, child1,
child2);
```

JS Syntax Using Factories

- We can use factories to simplify the component use from JS:

```
var Factory = React.createFactory(ComponentClass);
...
var root = Factory({ custom: 'prop' });
ReactDOM.render(root, document.getElementById('example'));
```

- For standard components like there are factories built-in:

```
var root = React.DOM.ul({ className: 'my-list' },
            React.DOM.li(null, 'Text Content')
          );
```

JS Syntax in Depth

- Since **JSX is JavaScript**, identifiers such as **class** and **for** are discouraged as XML attribute names. Instead, React DOM components expect DOM property names like **className** and **htmlFor**, respectively.

```
var myDivElement = <div className="foo" />;
ReactDOM.render(myDivElement,
                document.getElementById('example'));
```

- To render it use **Uppercase variable** → comp. displayName:

```
var MyComponent = React.createClass({/*...*/});
var myElm = <MyComponent someProperty={true} />;
ReactDOM.render(myElm, document.getElementById('example'));
```

JavaScript Expressions

- Attribute Expressions:

```
var person = <Person name=
                    {window.isLoggedIn ? window.name : “”} />;
```

- Boolean Attributes:

```
<input type="button" disabled />;
<input type="button" disabled={true} />;
```

- Child Expressions:

```
var content = <Container>
    {window.isLoggedIn ? <Nav /> : <Login />}
</Container>;
```

JSX Spread Attributes

- Mutating props is bad – should be treated as immutable
- Spread Attributes:

```
var props = {};
props.foo = x;
props.bar = y;
var component = <Component {...props} />;
```

- Order is important – property value overriding:

```
var props = { foo: 'default' };
var component = <Component {...props}  foo={'override'} />;
console.log(component.props.foo); // 'override'
```

HTML Entities in JSX

- Double escaping (all content is escaped by default – XSS):

```
<div>First &middot; Next</div> - OK
<div>{'First &middot; Next'}</div> - Double escaped
```

- Solution 1: type (and save) it in UTF-8:

```
<div>{'First · Next'}</div>
```

- Solution 2: use Unicode

```
<div>{'First \u00b7 Next'}</div>
<div>{'First ' + String.fromCharCode(183) + ' Next'}</div>
```

- Solution 3: use mixed arrays with strings and JSX elements:

```
<div>{['First ', <span key="middot">&middot;</span>, 'Next']}</div>
```

- Solution 4 (last resort): type (and save) it in UTF-8:

```
<div dangerouslySetInnerHTML={{__html: 'First &middot; Next'}} />
```

HTML Entities in JSX

- If you pass properties to native HTML elements that do not exist in the HTML specification, React will not render them.
- Custom attributes - should be prefixed with **data-** :

```
<div data-custom-attribute="foo" />
```

- Custom elements (with a hyphen in the tag name) support arbitrary attributes:

```
<x-my-component custom-attribute="foo" />
```

- Web Accessibility attributes starting with **aria-** are rendered:

```
<div aria-hidden={true} />
```

Immediately-Invoked Function Expressions

```
return (
  <section>
    <h1>Color</h1>
    <h3>Name</h3> <p>{this.state.color || "white"}</p>
    <h3>Hex</h3><p>
      {(() => {
        switch (this.state.color) {
          case "red":   return "#FF0000";
          case "green": return "#00FF00";
          default:      return "#FFFFFF";
        }
      })()}
    </p>
  </section>
);
```

React Component Lifecycle Callbacks (1)

- React components **lifecycle has 3 phases**:
  - **Mounting**: A component is being inserted into the DOM.
  - **Updating**: A component is being re-rendered to determine if the DOM should be updated.
  - **Unmounting**: A component being removed from the DOM.
- **Mounting** lifecycle callbacks:

getInitialState(): invoked before a component is mounted. Stateful components should implement this and return the initial state data.

componentWillMount(): invoked immediately before mounting

componentDidMount(): is invoked immediately after mounting occurs. Initialization that requires DOM nodes should go here.

React Component Lifecycle Callbacks (2)

- **Updating** lifecycle callbacks:

componentWillReceiveProps(object nextProps) – invoked when a mounted component receives new props. This method should be used to compare **this.props** and **nextProps** to perform state transitions using **this.setState()**.

shouldComponentUpdate(object nextProps, object nextState): boolean – invoked when a component decides whether to update - optimization comparing **this.props** with **nextProps** and **this.state** with **nextState** and return **false** if React should skip updating.

componentWillUpdate(object nextProps, object nextState) – invoked immediately before updating – do not call **this.setState()**.

componentDidUpdate(object prevProps, object prevState) – invoked immediately after updating occurs.

React Component Lifecycle Callbacks (3)

- **Unmounting** lifecycle callbacks:

componentWillUnmount() – invoked immediately before a component is unmounted and destroyed. Cleanup should go here.

- **Mounted** composite components also support:

component.forceUpdate() – can be invoked on any mounted component when you know that some deeper aspect of the component's state has changed without using **this.setState()**.

React Hooks – New in React 16!

[https://reactjs.org/docs/hooks-intro.html]

- **Hooks** are a new addition in React 16.8. They let you use state and other React features without writing a class.
- **Basic Hooks**
  - **useState:** const [state, setState] = useState(initialState);
  - **useEffect**: useEffect(() => {

    const subscription = props.source.subscribe();

    return () => { subscription.unsubscribe() }; });
  - **useContext** – allows to access resources appliucation wide
- **Additional Hooks** – useReducer, useCallback, useMemo, useRef, useImperativeHandle, useLayoutEffect, useDebugValue – will be discussed later during the course

React Hooks Example

[https://github.com/iproduct/course-node-express-react/tree/master/04-mybooks-lab4]

```
const GOOLE_BOOKS_API_BASE = "https://www.googleapis.com/books/v1/volumes?q=";

function App() {
  const [books, setBooks] = useState(mockBooks);
  return (
    <React.Fragment>
    <Nav searchBooks={onSearchBooks} />
    <div className="section no-pad-bot" id="index-banner">
      <div className="container">
      <Header />
      <BookList books={books} />
      </div>
    </div>
    <Footer />
    </React.Fragment>
  );

  async function onSearchBooks(searchText) {
    const booksResp = await fetch(GOOLE_BOOKS_API_BASE + encodeURIComponent(searchText));
    const booksFound = await booksResp.json();
    console.log(booksFound.items);
    setBooks(booksFound.items.map(gbook => ({
      'id': gbook.id,
      'title': gbook.volumeInfo.title,
      'subtitle': gbook.volumeInfo.subtitle,
      'frontPage': gbook.volumeInfo.imageLinks && gbook.volumeInfo.imageLinks.thumbnail
    })));
  }
}
```

Component Properties Validation (1)

```
React.createClass({
  propTypes: {
    // Optional basic JS type properties
    optionalArray: React.PropTypes.array,
    optionalBool: React.PropTypes.bool,
    optionalFunc: React.PropTypes.func,
    optionalNumber: React.PropTypes.number,
    optionalObject: React.PropTypes.object,
    optionalString: React.PropTypes.string,
    optionalSymbol: React.PropTypes.symbol,
```

Component Properties Validation (2)

```
// Anything that can be rendered: numbers, strings, elements or
// an array (or fragment) containing these types.
optionalNode: React.PropTypes.node,

// A React element.
optionalElement: React.PropTypes.element,

// You can also declare that a prop is an instance of a class.
optionalMessage: React.PropTypes.instanceOf(Message),
```

Component Properties Validation (3)

```
// You can ensure that your prop is limited to specific enum.
optionalEnum: React.PropTypes.oneOf(['News', 'Photos']),

// An object that could be one of many types
optionalUnion: React.PropTypes.oneOfType([
  React.PropTypes.string,
  React.PropTypes.number,
  React.PropTypes.instanceOf(Message)
]),
```

Component Properties Validation (4)

```
 // An array of a certain type
optArray: React.PropTypes.arrayOf(React.PropTypes.number),

// An object with property values of a certain type

optObject:React.PropTypes.objectOf(React.PropTypes.number),

// An object taking on a particular shape
optionalObjectWithShape: React.PropTypes.shape({
  color: React.PropTypes.string,
  fontSize: React.PropTypes.number
}),
```

Component Properties Validation (5)

```
 // You can chain any of the above with `isRequired`
requiredFunc: React.PropTypes.func.isRequired,

// A required value of any data type
requiredAny: React.PropTypes.any.isRequired,

// You can also specify a custom validator => return an Error
customProp: function(props, propName, componentName) {
  if (!/matchme/.test(props[propName])) {
    return new Error('Invalid prop `' + propName + '` supplied to' +
      ' `' + componentName + '`. Validation failed.'
    ); }}   /* ... */
```

Events in React

- **SyntheticEvent(s)** - event handlers are passed instances of SyntheticEvent – cross-browser wrapper around native events
- Same interface: **stopPropagation()**, **preventDefault()**
- **Event pooling** – all SyntheticEvent(s) are pooled = objects will be reused and all properties will be nullified after the event callback has been invoked (performance) -not for async access:

  Example: https://facebook.github.io/react/docs/events.html
- If you want event to be persistent, call: **event.persist()**
- Event types: **Clipboard, Composition, Keyboard, Focus, Form, Mouse, Selection, Touch, UI Events, Wheel, Media, Image, Animation, Transition**

Component Ownership

- **Multiple Components** – allow separation of concerns and reusability
- **Ownwrship** – an owner is the component that **sets the props** of owned components.
- When a component **X** is created in component **Y**'s **render()** method, it is said that **X** is owned by **Y**.
- Only defined for React components – different from parent-child DOM relationship.
- **Child Reconciliation** – the process by which React updates the DOM with each new render pass. In general, children are reconciled according to the order in which they are rendered.

Reconciliation Example

- // Render Pass 1

```
<Card>
    <p>Paragraph 1</p>
    <p>Paragraph 2</p>
</Card>
```

- // Render Pass 2

```
<Card>
    <p>Paragraph 2</p>
</Card>
```

Stateful Children Reconciliation – Keys

```
var ListItemWrapper = React.createClass({
  render: function() {
    return <li>{this.props.data.text}</li>;
  }
});
var MyComponent = React.createClass({
  render: function() {
    return (
      <ul>
        {this.props.results.map(function(result) {
          return <ListItemWrapper key={result.id} data={result}/>;
        })}
      </ul>
    );  }
});
```

React.Children Utilities

- **React.Children.map(object children, function fn [, object thisArg]): array** – invoke **fn** on every immediate child contained within **children** with **this** set to **thisArg**
- **React.Children.forEach(object children, function fn [, object thisArg])** – same as **map**, but does not return an array
- **React.Children.count(object children): number** - returns children count
- **React.Children.only(object children): object** – returns the only child in children. Throws otherwise
- **React.Children.toArray(object children): array** – returns the children as a flat array with keys assigned to each child

Transferring Props

```
function FancyCheckbox(props) {
  let { checked, ...other } = props;
  let fancyClass = checked ? 'FancyChecked' : 'FancyUnchecked';
  // `other` contains { onClick: console.log } but not the checked property
  return (
    <div {...other} className={fancyClass} />
  );
}
ReactDOM.render(
  <FancyCheckbox checked={true} onClick={console.log.bind(console)}>
    Hello world!
  </FancyCheckbox>,
  document.getElementById('example')
);
```

Forms in React – Controlled Components

- **Interactive Props** - form components support a few props that are affected via user interactions:
  - value - supported by <input> and <textarea> components
  - checked - supported by <input> of type checkbox or radio
  - selected - supported by <option> components
- Above form components allow listening for changes by setting a callback to the **onChange** prop:

```
handleAuthorChange(e) { this.setState({author: e.target.value}); }
<input type="text" value={this.state.author} placeholder="Your name"
    onChange={this.handleAuthorChange}/>
```

- **Controlled component** does not maintain its own internal state – the component renders purely based on **props**

Refs to Components

- **Refs (references)** – allow to find the **DOM markup** rendered by a component, and invoke methods on **component instances** returned from **render()**
- Example uses: absolute positioning, using React components in larger non-React applications, transition existing code to React.

```
var myComponentInstanceRef =
        ReactDOM.render(<MyComp />, myContainer);
myComponentInstanceRef.doSomething();
```

- `ReactDOM.findDOMNode(componentInstance)` – this function will return the DOM node belonging to the outermost HTML element returned by render.

The ref Callback Attribute

```
render: function() {
    return (
      <TextInput ref={ function(input) {
          if (input != null) {
            input.focus();
          }
      }} />
    );
},
```

- **OR using ES6 => :**

```
render: function() {
    return <TextInput ref={(c) => this._input = c} />;
},
componentDidMount: function() {
    this._input.focus();
},
```

The ref String Attribute

- Assign a **ref** attribute to anything returned from **render** such as:

```
<input ref="myInput" />
```

- In some other code (typically event handler code), access the backing instance via **this.refs** as in:

```
var input = this.refs.myInput; // better this.refs['myInput'];
var inputValue = input.value;
var inputRect = input.getBoundingClientRect();
input.focus();
```

The ref String Attribute - Example

```
var MyComponent = React.createClass({
  handleClick: function() {
    React.findDOMNode(this.refs.myTextInput).focus();
  },
  render: function() {
    return (
      <div>
        <input type="text" ref="myTextInput" />
        <input type="button" value="Focus the text input"
               onClick={this.handleClick} />
      </div>
    );
  }
});
```

Thanks for Your Attention!

Questions?